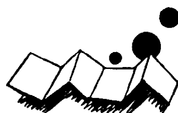
解　説



2. 方式・機能・論理設計における CAD

2.5 方式・機能・論理設計の検証†

丸 山 文 宏†† 上 原 貴 夫††

1. はじめに

論理装置が LSI 化され，チップの集積度が上がるにつれて，設計の誤りをどうやってなくすかがますます重大な問題となってきている．何万ゲートというチップの上に設計ミスがあれば，プリント板のように配線変更はできず，そのチップをつくり直さなければならないことも起こり得る．膨大な開発費を投じたチップのつくり直しは極めて高価なものとなる．したがって，設計時に誤りを零にすること，あるいは少し譲って零に近くすることは，最近の論理装置開発において大変重要なこととなっている．この問題に対してひとつの解答を与えるものがこれから述べる設計の検証 (Formal Verification) である．

設計のチェックを行うのに最もよく用いられる方法はシミュレーションであろう．特に，設計チェックの初期段階においては，シミュレーションが大きな威力を発揮する．設計者はシミュレーションの結果を追い，洗い出された設計ミスを直していく．しかし，シミュレーションをくり返して誤りが見つからなくなったとしても，すべての場合を当たったのでない限り，それは誤りが存在しないことを保証するものではない．つまり，シミュレーションは，設計ミスの発見には有効だが，誤りを含まない正しい設計であると保証することはできない．本稿では，シミュレーションによらずに設計の正しさを検証する手法について述べる．

2. 形式的推論(Formal Reasoning)による検証

シミュレーションによらずに設計の検証を行う手法として，形式的推論により設計の正しさを証明するものが提案されている．論理 (Logic) が形式的推論において中心的な役割を果たす．例えば，論理回路を論理を用いて正確に記述しておけば，仕様を論理で表現することにより，基本的には検証は定理の証明に帰着される．最も簡単なものには図-1 に示す組合せ回路の検証がある．図-1 の2つの回路がいずれも排他的論理和 (exclusive OR) を実現していることは，排他的論理和と等価な論理式の判定に帰着される．

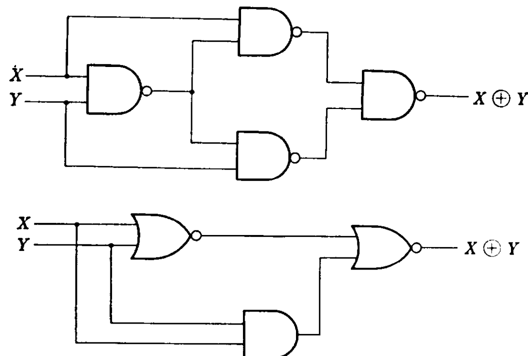


図-1　排他的論理和 (exclusive-or) の2つの実現例

Wagner は，スタンフォード大学で開発されたプルーフチェッカ (FOL)* を用いた検証について報告している[1]．260 ステップに及ぶ8ビット乗算器の検証は見事であるが，その問題点は，プルーフチェッカの助けを借りて自分で証明を構成しなければならないことである．設計者に自ら証明を構成することを要求するのは，CAD の観点から見て望ましくない．CAD の目指すものは自動あるいは半自動の検証であろう．Wagner の方法を自動化しようとする試みもあるが，加算器，シフタ等特殊な回路に限られている[2]．

論理を用いた自動証明という考え方が何を生み出したかを見る前に，まず，検証すべき仕様としてはどのようなものが考えられるか，どのようなものが表現できなくてはならないかについて考える．

† Verification of System, Functional, Logic Design by Fumihiro MARUYAMA and Takao UEHARA (Software Laboratory, Fujitsu Laboratories Ltd.).

†† (株)富士通研究所ソフトウェア研究部第一研究室

* FOL はインタラクティブに証明を行うシステムであり，各ステップごとにユーザは使うべき公理を指定する．

2.1 仕様の記述

第一に設計者が検証したい仕様は,「ある条件が成り立っていれば必ず別のある条件も成立している」ことであろう. 例えば, あるレジスタをフェッチする条件が成り立つ時にはそのレジスタには正しい内容がはいっていなければならない等. これは次の論理式で表現できる.

$$C1 \supset C2 \qquad (1)$$

($\supset$ は implication〔含意〕)

ただし, $C1$ と $C2$ (どちらも論理式) が上の仕様の2つの条件にそれぞれ対応する. (1)は, $C1$ を常に真である論理式とすることにより,「ある条件が常に成立する」ことも特別な場合として含む. 前に述べた組合せ回路の検証もこのカテゴリに含まれる.

次に,「ある条件が成り立てばやがて別のある条件も成立する」ことの検証も有用であろう. この仕様には, 2つの条件が成立するタイミングにずれがある場合も含まれる. 例えば, 外部からのある信号が ON になれば, やがてあるレジスタには外部からの正しいデータが取り込まれている等. これは通常の論理では表現できない. そこでテンポラル・ロジックを導入する.

以下で引用するテンポラル・ロジックは, Pnueli が発展させ[3], Hailpern が並列プロセスの検証に用いた[4] ものである. テンポラル・ロジックは通常の論理に次の3つのオペレータ (modal operator) を付け加えた拡張である.

□ henceforth (これ以後ずっと)

◇ eventually (やがて)

○ next (次のタイミングで)

□Pは, Pは現在真でありずっと真であり続けることを意味する. ◇Pは, Pが現在真であるかまたは将来真になることを意味する. ○Pとは次のタイミングでPが真になることを示す.

上に述べた仕様はテンポラル・ロジックを用いて次のように表現される.

$$\Box(C1 \supset \Diamond C2) \qquad (2)$$

ただし, $C1$ と $C2$ (どちらも論理式) が仕様の2つの条件に対応する. $C1$ を常に真である論理式とすることにより,「ある条件はどのような状況のもとでもやがて真になる」ことも表現できる.

最後に考察する仕様は,「ある条件がいったん真になると別のある条件が成立するまで真であり続ける」ことである. 例えば, 外部からの信号に対してきちんと応答信号を上げていること等. これは次のように表現できる.

$$\Box(C1 \supset (C2 \vee \bigcirc(C1 \vee C2))) \qquad (3)^{*}$$

すなわち, $C1$ が真なら, $C2$ がすでに真か (これで OK), あるいは次のタイミングで $C2$ が真となるか $C1$ がやはり真である.

以上の仕様について, どのように自動証明を実行するのかを次に考察する.

2.2 逆方向推論による検証

先に, Wagner の方法では設計者に証明を構成することを要求し自動化が難しいと述べた. したがって, 自動証明のアプローチとしては, 構成的な証明ではなく分析的な分解証明が適している. すなわち, 目標 (ゴール) から出発して, その分解を繰り返すことにより最終的に自明な条件に帰着させる方法により自動証明を実現することを考える.

以下に紹介する手法では, ゴールとして仕様の否定を取り, 背理法による証明を実行している. 仕様の否定 (仕様が成り立たない状況) から出発し矛盾を導き出そうとする. この時行われる分解では因果関係が追跡される. つまり, 仕様の否定が成り立つ (仕様が満たされない) ならばその前のタイミングではどのような条件が成立していなくてはならないか (原因), を因果関係の追跡によって見いだす. その条件から矛盾が生じるならば仕様は必ず成り立っていることになり検証は成功である. そうでなければ更にその前のタイミングに戻る. このように, 原因を求めて過去にさかのぼる逆方向推論によって検証ができる.

具体的なシステムの例として, 次章で DDL ベリファイア[5)-8)]について解説する. 藤田らは Prolog による検証システムを構築した. このシステムでは, テンポラル・ロジックで書かれた仕様ならば何でも受け付ける. これについては他の文献[9)-11)]を参照されたい.

3. DDL ベリファイア

DDL ベリファイアは, ハードウェア記述言語 DDL で記述された設計を検証するソフトウェア・システムである. まず, DDL について説明する.

3.1 DDL

DDL は, Dietmeyer と Duley によって開発されたレジスタ・トランスファ・レベルのハードウェア記述言語であり, ステートマシン記述, 系統的なトランスレーション・アルゴリズム等の特徴を持つ[12)-14)].

---

* U (until) オペレータを用いれば $\Box(C1 \supset (C1 \, U \, C2))$ と書ける.

表-1　DDL（富士通版）の記法

記　号	意　　　味
~	否　定 (not)
&	論理積 (and)
¦	論理和 (or)
:=	等　号
=	結　合
<-	転　送
->	遷　移
¦*　*¦	IF 節
;	ELSE
/*　*/	コメント
$	オートマトン名による修飾

表-1 に富士通で使われている DDL の記法を示す．

DDL では論理回路のまとまった単位をファシリティと呼ぶ．例えば，メモリ，レジスタ，ターミナル（信号線）等である．ファシリティはオートマトンと呼ばれる独立した制御単位に分かれる．各オートマトンはステートマシンと見なされ，各サイクル（クロック・パルスで区切られた時間単位）においてちょうどひとつのステートをとる．次のサイクルのステートを指定するのが遷移オペレーションである：

　　-> SID　(SID はステートの識別子)

結合オペレーション，転送オペレーションはそれぞれターミナル，レジスタ（あるいはメモリ）に作用する．次の結合オペレーション

　　ID=BE

は，右辺の論理式 BE で指定された組合せ回路の出力端子が左辺の識別子 ID で指定されたターミナルにつながれることを意味する．DDL においては，組合せ回路は遅延のない理想的なものと考えられている．ターミナルに結合される値が指定されていない時にはそのターミナルの値は0とする．一方，次の転送オペレーション

　　ID<-BE

は，レジスタ（あるいはメモリ）ID に論理式 BE で指定されたネットワークから値が転送されることを示す．結合オペレーションは時間遅れのないオペレーションであるが，レジスタ（あるいはメモリ）への転送には，ステート遷移と同様，1サイクルかかる．

　　|BE| OP1；OP0.

は IF-THEN-ELSE 条件付オペレーションを定義する．BE の値が1ならば，一連のオペレーション OP1 が実行され，0ならば OP0 が実行される．この変形が IF-THEN 構文である：

|*C1*| T=A & B.
⋮
|*C2*| T=D.

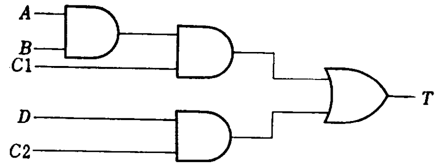


図-2　ターミナルと結合オペレーション

|*C1*| R<-A.
⋮
|*C2*| R<-B.

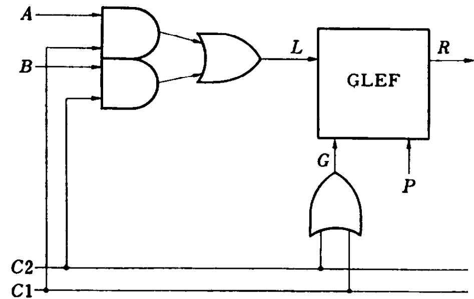


図-3　レジスタと転送オペレーション

　　|BE| OP1.

この場合には BE が0の時には何も実行されない．

図-2，3 はそれぞれ結合オペレーション，転送オペレーションを論理回路によるひとつの実現例とともに示している．図-4 には簡単な計算機のモデルの DDL 記述を示す．

DDL トランスレータは，DDL 記述から「因果関係テーブル」を生成する．因果関係テーブルには，すべてのオペレーション（結合，転送，遷移）が各ターミナル，レジスタ（あるいはメモリ），ステートごとにまとめられ，その実行条件とともに編集されている．図-5 にその一例を示す．

3.2 因果関係

図-3 の例では，条件 $C1$ は A の内容をレジスタ R に転送する．その時の A の値が次のサイクルの R の出力を決定する．因果関係の追跡とは，大ざっぱに言って，結果（この場合，R の出力）から出発して，原因（1サイクル前の A の値と転送条件 $C1$）に戻ることを言う．ただし，常に1つ前のサイクルに戻るわけではない．ターミナルに関しては，組合せ回路の出力信号から入力信号にさかのぼる．因果関係テーブルが

```
⟨SYSTEM⟩ SAMPLE:
  ⟨TIME⟩ P (100).    /* CLOCK */
    ⟨ENTRANCE⟩ DATAAV, CHTOMEM (16).  /*TERMINALS FROM OUTSIDE */
    ⟨REGISTER⟩ OP (16), INVALID, CHSTORE.
    ⟨TERMINAL⟩ MEMAV, BUS (16), MOVEIN, CPUTOCH, MEMTOCH, FETCH.
    ⟨AUTOMATON⟩ CPU: P: /* AUTOMATON CPU IS CONTROLLED BY CLOCK P */
      ⟨REGISTER⟩ BS (16).
      ⟨STATES⟩
        CPUS1:          /* STATE CPUS1 */
          ¦* MEMAV *¦
           ¦* OP (0:1)=0 *¦       /* CHANNEL STORE */
             INVALID <-1, CPUTCOH=1, CHSTORE <-0, -> CPUS1.,
           ¦* OP (0:1)=1 *¦
            ¦* INVALID *¦     /* MOVE-IN */
             ¦* CHSTORE *¦
               MOVEIN=1, -> CPUS2
             ;  -> CPUS1.
            ; FETCH=1, -> CPUS1.., /* FETCH OPERATION */
            /* DETAIL OF FETCH OPERATION IS OMITTED */
           ¦* ~(OP (0:1)=0 ¦ OP (0:1)=1) *¦
            /* OPERATIONS ARE NOT DESCRIBED EXCEPT STORE AND FETCH */
            -> CPUS1.
          ; -> CPUS1..
        CPUS2:          /* STATE CPUS2 */
          INVALID <-0, BS <-BUS, -> CPUS1.
      ⟨END⟩.
    ⟨END⟩ CPU.
    ⟨AUTOMATON⟩ MEMORY: P: /* AUTOMATON MEMORY IS CONTROLLED BY CLOCK P */
      ⟨REGISTER⟩ M (16).
      ⟨STATES⟩
        MEMS1:          /* STATE MEMS1 */
          MEMAV=1,
          ¦* MOVEIN *¦ -> MEMS2
          ; ¦* DATAAV & INVALID *¦
             M <-CHTOMEM, MEMTOCH=1, CHSTORE <-1..
           -> MEMS1..
        MEMS2:          /* STATE MEMS2 */
          BUS=M, -> MEMS1.
      ⟨END⟩.
    ⟨END⟩ MEMORY.
⟨END⟩ SAMPLE.
```

図-4 DDL 記述

そのための情報を提供する．**図-5**(a) から，ターミナル T が ON であるのは

$$(A \ \& \ B \ \& \ C1)|(D \ \& \ C2) \qquad (4)$$

が真の時でありその時に限ることがわかる．(4)を T (正確には T=1) の必要十分条件と呼ぶ．因果関係の追跡は，実際には「置換」と呼ばれる論理式上の操作によって実現される．ターミナル置換は，ターミナルを含む部分論理式をその必要十分条件で置き換えることである．例えば，(4)が T の代わりに置き換えられる．

図-5(b) のレジスタ R の1サイクル前における必要十分条件は

$$(A \ \& \ C1)|(B \ \& \ C2)|(R \ \& \ \sim(C1|C2)) \qquad (5)$$

である．明らかに，レジスタ R が ON であるのは，1サイクル前で(5)が真であった場合でありまたその場合に限る．レジスタ置換とは，レジスタ (あるいはメモリ) のみから成る部分論理式をその1サイクル前における必要十分条件で置き換えることである．例えば，(5)がレジスタ R の代わりに置き換えられる．

ステート置換は，ステートをその1サイクル前における必要十分条件 (因果関係テーブルに明確に示されている) で置き換える．例えば，**図-5**(c)の ST1 の代わりには

$$(\mathrm{ST1} \ \& \ \sim C1)|(\mathrm{ST2} \ \& \ C2)$$

T (0 : 0) 1 BIT TERMINAL		
RANGE	SOURCE	CONNECTION CONDITION
(0)	A & B	C1
	D	C2

(a) terminal

R (0 : 0) 1 BIT REGISTER		
RANGE	SOURCE	TRANSFER CONDITION
(0)	A	C1
	B	C2

(b) register

PRESENT STATE	PREVIOUS STATE	CONDITION
ST1	ST1	~C1
	ST2	C2
ST2	ST1	C1
	ST2	~C2

(c) state

図-5 因果関係テーブル
(a), (b) はそれぞれ図-2, 3 に対応.

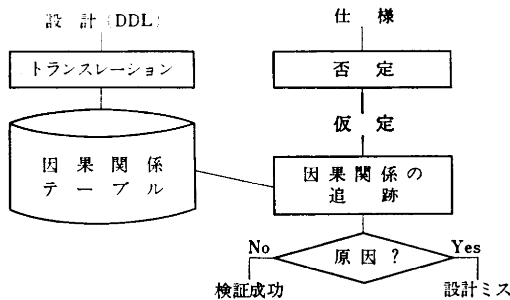


図-6 DDL ベリファイアの構成

が置き換えられる.

3.3 アルゴリズム

図-6 に DDL ベリファイアの構成を示す. 因果関係の追跡は仕様が満たされないと仮定するところから出発して原因を見つけ出そうとする. もし本当に起こり得る原因が見つからなければ, 仕様の反例は存在せず, 仕様は検証されたことになる. そうでなければ, この設計でなぜ仕様が満たされないかをその原因が示すことになる.

以下に因果関係追跡のアルゴリズムを示す.

[ステップ1] 論理式にターミナルが含まれている限りターミナル置換を繰り返す. もし得られた論理式が常に偽ならば検証成功. そうでなければ [ステップ2] へ.

[ステップ2] DDL 記述の中に値をセットするオペレーションが現れないファシリティは任意の値を取り得るとみなす. [ステップ3] へ.

[ステップ3] 論理式中のすべてのステートとすべてのレジスタだけから成る部分論理式についてステート置換, レジスタ置換を実行する. もし得られた論理式が常に偽ならば検証成功. そうでなければ [ステップ1] へ.

得られた論理式が常に偽かどうかを調べるために用いられている公理は次のようなものである:

$A \ \& \sim A \equiv \phi$,

(ϕ は常に偽である論理式. 以下同様)

$B \neq B \equiv \phi$,

$(B := \mathrm{v1}) \ \& \ (B := \mathrm{v2}) \equiv \phi \quad (\mathrm{v1} \neq \mathrm{v2})$,

$\mathrm{ST\,1} \ \& \ \mathrm{ST\,2} \equiv \phi$. (ST 1, ST 2 は同一オートマトンの2つの異なるステート)

最後の公理は DDL に固有のものである.

[ステップ2] はもともと外部からの入力信号 (エントランスと呼ばれる) を念頭において考えられた. エントランスには結合オペレーションがない. ベリファイアはエントランスによらずに検証を行おうとする. 例えば, 部分論理式として

$$E := 1 \quad (E \text{ はエントランス}) \qquad (6)$$

が含まれている時, ベリファイアは(6)を常に真である論理式に変える. この処理は論理式をその必要条件に変更する. そのために正しい設計でも検証が成功しない可能性もある. しかし, 誤った設計が正しいと検証されてしまうことはない. また, 論理式を小さくするのに役立つ. 必要十分条件の追跡の結果, しばしば巨大な論理式が生じ, しかもその多くの部分は検証には無関係である. ベリファイアは設計者に (検証に関係あると思われる) レジスタを指定させ, それ以外のレジスタに対し上の処理を適用する.

論理式Cに [ステップ1]~[ステップ3] を適用して得られる論理式を $\mathrm{np}(C)$ と書く. もし C が真ならば $\mathrm{np}(C)$ は1サイクル前において真でなくてはならない. 更に, $\mathrm{np}^i(C)$ $(i=0, 1, 2, 3, \cdots)$ を次のように定義する:

$$\mathrm{np}^0(C) = C,$$

$$\mathrm{np}^i(C) = \mathrm{np}(\mathrm{np}^{i-1}(C)). \quad (i>0)$$

もし論理式Cが真ならば $\mathrm{np}^i(C)$ は i サイクル前において真でなくてはならない. もし $\mathrm{np}^i(\sim P)$ が常に偽

```
***** DDL VERIFIER *****

***** YOU CAN SELECT ONE OF THE FOLLOWING THREE CONDITIONAL ASSERTIONS *****
  A : (CONDITION1) ALWAYS IMPLIES (CONDITION 2).
  B : IF (CONDITION1) IS TRUE, (CONDITION 2) WILL EVENTUALLY BECOME TRUE.
  C : ONCE (CONDITION1) BECOMES TRUE, IT WILL REMAIN TRUE UNTIL (CONDITION 2) BECOMES
      TRUE.

***** ENTER THE TYPE YOU WANT TO VERIFY. A/B/C/ ? (FOR HELP)/Q (TO QUIT)
:
A

THE FOLLOWING TYPE OF ASSERTION HAS BEEN CHOSEN:
     (Premise) ALWAYS IMPLIES (Conclusion).

ENTER PREMISE
:
FETCH

ENTER CONCLUSION
:
CPU$BS=MEMORY$M

***** SPECIFY REGISTERS TO BE RETRACED
:
INVALID

:
CPU$BS

:
MEMORY$M

:

***** ENTER CONDITIONS UNDER WHICH THE VERIFICATION SHOULD BE DONE
:

***** ENTER INITIAL CONDITIONS
:
INVALID

:

MEMORY$MEMS1 & CPU$CPUS1 & ~INVALID & ~(CPU$BS=MEMORY$M) & ~(MEMORY$M=CPU$BS)
RETRACING TO1 CYCLES BEFORE

MEMORY$MEMS1 & CPU$CPUS2 & INVALID
RETRACING TO2 CYCLES BEFORE
RETRACING TO1 CYCLES BEFORE

MEMORY$MEMS1 & CPU$CPUS2 & ~(MEMORY$M=ØØØØØØØØØØØØØØØØ)
RETRACING TO2 CYCLES BEFORE
RETRACING TO1 CYCLES BEFORE

MEMORY$MEMS2 & CPU$CPUS1 & ~INVALID & ~(CPU$BS=MEMORY$M) & ~(MEMORY$M=CPU$BS)
RETRACING TO2 CYCLES BEFORE
RETRACING TO1 CYCLES BEFORE

***** FETCH ALWAYS IMPLIES CPU$BS=MEMORY$M *****
```

図-7 つづく

```
***** ENTER THE TYPE YOU WANT TO VERIFY. A/B/C/? (FOR HELP)/Q (TO QUIT)
:
B

THE FOLLOWING TYPE OF ASSERTION HAS BEEN CHOSEN:
     IF (CONDITION 1) IS TRUE, (CONDITION 2) WILL EVENTUALLY BECOME TRUE.

ENTER CONDITION 1
:

ENTER CONDITION 2
:
MEMAV

WOULD YOU LIKE TO SPECIFY HOW SOON CONDITION 2 IS SUPPOSED TO BECOME TRUE? Y/N
:
N

ENTER CONDITIONS UNDER WHICH THE VERIFICATION SHOULD BE DONE
:

***** MEMAV WILL BECOME TRUE WITHIN 1 CYCLES.

***** ENTER THE TYPE YOU MANT TO VERIFY. A/B/C/? (FOR HELP)/Q (TO QUIT)
:
Q
```

図-7 DDL ベリファイアによる検証例

であることが示されれば，$\sim P$が真になるという仮定が否定され，Pが常に真であることが検証される．

これまでは検証成功の場合についてのみ述べてきた．ここで検証が失敗する場合にも触れておきたい．例えば $\mathrm{np}^i(\sim P)$ (i はある正整数) が $\sim P$より弱い条件であることがわかったと仮定する．この手法では，$\sim P$ が常に偽であることを証明するために $\mathrm{np}^i(\sim P)$ が常に偽であることを示す必要がある．しかし，この場合，$\sim P\equiv\phi$ を示すことより $\mathrm{np}^i(\sim P)\equiv\phi$ を示す方が難しい．したがって，この手法で $\sim P\equiv\phi$ を検証することはできない．次のような診断メッセージが出力される：もし条件 $\mathrm{np}^i(\sim P)$ が真となれば条件 P は i サイクル後に偽となる可能性がある．

3.4 検証のタイプ

ここでは，前に述べた3種類の仕様をベリファイアがどのように扱うのかについて説明する．(1)の否定は

$$C1\ \&\ \sim C2 \tag{7}$$

だから，ベリファイアは $C1$ と $C2$ を入力して(7)を構成し，**3.3** のアルゴリズムを適用する．(2)の否定は

$$\Diamond\ (C1\ \&\ \Box \sim C2) \tag{8}$$

である（テンポラル・ロジックの公理 $\sim\Box P=\Diamond\sim P$ 及び $\sim\Diamond P=\Box\sim P$ より)．これは

$$C1\ \&\ \sim C2\ \&\ \mathrm{np}(\sim C2)\ \&\ \mathrm{np}^2(\sim C2)\ \&\cdots$$

が真となることを意味する．ベリファイアは，i を0を初期値として1ずつ増しながら

$$C1\ \&\ \sim C2\ \&\ \mathrm{np}(\sim C2)\ \&\cdots\&\ \mathrm{np}^i(\sim C2)\equiv\phi$$

が成り立つかどうかを調べる．このような i が見つかれば検証は成功である．更に次のことが言える：$C1$ が成り立てば i サイクル以内に $C2$ が成立する．(3)の否定は

$$\Diamond\ (C1\ \&\ \sim C2\ \&\ \bigcirc\ (\sim C1\ \&\ \sim C2))$$

である（テンポラル・ロジックの公理 $\sim\bigcirc P=\bigcirc\sim P$ による)．

これは

$$C1\ \&\ \sim C2\ \&\ \mathrm{np}(\sim C1\ \&\ \sim C2) \tag{9}$$

が真となることを意味する．ベリファイアは，(9)$\equiv\phi$ を導こうとする．もし成功すれば(3)が検証されたことになる．

3.5 検 証 例

図-7 には DDL ベリファイアによる図-4 の設計に関する2つの検証例を示す．最初の例では，信号 FETCH が ON の時にはいつもバッファ（レジスタ）

CPU＄BS とメモリ MEMORY＄M の内容が同一であることを検証している．信号 FETCH は CPU がバッファの内容を取り出そうとしていることを示す信号であり，最新のデータは常にメモリに格納されているから，これは CPU が常に最新のデータを取り出すことを検証するものである．設計者は3つのレジスタ，INVALID, CPU＄BS, MEMORY＄M を指定した．その他のレジスタには 3.3 のアルゴリズムの［ステップ2］が適用される．特定の条件がいつも成立していると仮定してそのもとで検証を行うこともできるが，ここでは何も指定されなかった．初期条件としては INVALID (つまり INVALID=1) が指定された．これは，このハードウェアが立ち上げられる時にはいつもレジスタ INVALID は ON にセットされることを意味する．ちなみに，INVALID はバッファの内容が無効であることを示すフラグである．ベリファイアは

FETCH & ～(CPU＄BS=MEMORY＄M)

から出発して因果関係の追跡を行う．検証は成功した．ただし，指定された初期条件 (INVALID=1) はどこまでさかのぼっても INVALID=0 であるようなパスを除去するのに用いられた．

次の例は，どの時点からでもターミナル MEMAV はやがて ON になることの検証である．すなわち，メモリはいつもそのうちアクセス可能となることを検証する．設計者は "CONDITION 1" として常に真である論理式を入力した (単に "return" を入力することにより)．設計者は他の情報は入力しなかった．検証の結果は，どの時点においても MEMAV は現在 ON であるかあるいは次のサイクルで ON になることが判明した．

DDL ベリファイアは LISP マシンと汎用コンピュータをつなぐアダプタの設計の検証に適用された．アダプタの論理設計は100ページ余りの仕様書（日本語）に従って書かれた約1300行の DDL 記述である．大型コンピュータの上でDDL ベリファイアを走らせたところ，各検証を数分以下で実行できた．

4. その他の検証手法

ソフトウェアにおいてはプログラムの正当性，すなわち，プログラムが停止し，仕様に記述されている通りの振舞をすること，の検証について研究がなされてきた．ハードウェア設計の検証にこの成果を利用しようとする提案もある．例えば，Floyd の帰納的表明法の拡張が提案されている[15]．しかし，Floyd の方法は sequential なプログラムに対して適用されたものであり，並列動作の多いハードウェアの設計への適用には更に検討が必要であろう．

記号シミュレーションによるハードウェア設計の検証についても報告されている[16]．しかし，問題点は，シミュレーションが進むにつれてだんだん記号式が複雑になり解析が難しくなることである．斉藤らのシステムでは，データパスは記号値，制御信号は数値を用いるなど値の使いわけができ，これによって上に述べた問題に対処しようとしている[17]．また，このシステムでは，フレームとデモンを活用することにより，設計者の持つ知識をシミュレータに与えたり，ある素子間に成り立ってはならない関係を常時監視したりするなど，シミュレータに融通性を持たせることが可能である．

5. おわりに

以上，DDL ベリファイアを中心に設計の検証について概観した．触れることができなかった研究成果も多いと思われる．ただ，まだ確立したもののないこの分野におけるひとつの考え方を理解して頂ければ幸いである．

参考文献

1) Wagner, T. J.: Hardware Verification, Ph. D. dissertation, Comp. Sci. Dept., Rept. No. STAN-CS-77-632, Stanford Univ. (1977).
2) Wojcik, A. S.: Formal Design Verification of Digital Systems, 20th DAC, pp. 228-234 (1983).
3) Manna, Z. and Pnueli, A.: Verification of Concurrent Programs, Part 1: The Temporal Framework, Rept. No. STAN-CS-81-836, Stanford Univ. (1981).
4) Hailpern, B. T.: Verifying Concurrent Processes Using Temporal Logic, Tech. Rept. No. 195, Stanford Univ. (1980).
5) Maruyama, F. et al.: Hardware Verification and Design Error Diagnosis, FTCS-10, pp. 59-64 (1980).
6) Uehara, T. et al.: DDL Verifier, CHDL '81, pp. 51-64 (1981).
7) Maruyama F. et al.: A Verification Technique for Hardware Designs, 19th DAC, pp. 832-841 (1982).
8) Uehara, T. et al.: DDL Verifier and Temporal Logic, CHDL '83, pp. 91-102 (1983).
9) Fujita, M. et al.: Verification with Prolog

and Temporal Logic, IFIP 6th Computer Hardware Description Languages and their Applications (1983).

10) Fujita, M. et al.: Temporal Logic Based Hardware Description and its Verification with Prolog, New Generation Computing, Vol. 1, No. 2, Ohmsha and Springer-Verlag (1983).

11) Fujita, M. et al.: Efficient Verification Methods for Hardware Logic Design and their Implementation with Prolog, Proc. of the 2nd Logic Programming Conference of Japan, Tokyo, Japan (1984).

12) Duley, J.R. and Dietmeyer, D.L.: A Digital System Design Language (DDL), IEEE Trans. Comput., Vol. C-17, pp. 850-861 (1968).

13) Duley, J.R. and Dietmeyer, D.L.: Translation of a DDL Digital System Specification to Boolear Equation, IEEE Trans. Comput., Vol. C-18, pp. 305-313 (1969).

14) Kawato, N. et al.: Design and Verification of Large Scale Computer by Using DDL, 16th DAC, pp. 360-366 (1979).

15) Pitchumani, V. and Stabler, E.P.: An Inductive Assertion Method for Register Transfer Level Design Verification, IEEE Trans. Comput., Vol. C-32, pp. 1073-1080 (1983).

16) Carter, W.C. et al.: Symbolic Simulation for Correct Machine Design, 16th DAC, pp. 280-286 (1979).

17) 斎藤隆夫他：フレームとデモンによる CAD システム (FIDDLE), 電子通信学会電子装置設計技術研究会資料, 電子装置設計技術 9-1) (1981).

(昭和 59 年 6 月 4 日受付)